CS429: Computer Organization and Architecture

Instruction Set Architecture II

Warren Hunt, Jr. and Bill Young
Department of Computer Sciences
University of Texas at Austin

Last updated: October 1, 2014 at 08:36

Topics of this Slideset

- Assembly Programmer's Execution Model
- Accessing Information
- Registers
- Memory
- Arithmetic operations

BTW: We're through with Y86 for a while, and starting the x86. We'll come back to the Y86 later for pipelining.

IA32 Processors

x86 processors totally dominate the computer market.

Evolutionary Design

- Starting in 1978 with 8086
- Added more features over time.

Complex Instruction Set Computer (CISC)

- Still support many old, now obsolete, features.
- There are many different instructions with many different formats, but only a small subset are encountered with Linux programs.
- Hard to match performance of Reduced Instruction Set Computers (RISC), though Intel has done just that!

x86 Evolution: Programmer's View

Name	Date	Transistors
8086	**1978**	**29K**

- 16-bit processor. Basis for IBM PC and DOS.
- Limited to 1MB address space. DOS only gives you 640K

80286	**1982**	**134K**

- Added elaborate, but not very useful, addressing scheme.
- Basis for IBM PC-AT and Windows

386	**1985**	**275K**

- Extended to 32 bits. Added "flat addressing."
- Capable of running Unix.
- Linux/gcc uses no instructions introduced in later models.

x86 Evolution: Programmer's View

Name	Date	Transistors
486 Pro	**1989**	**1.9M**

- Added on chip floating point unit.

Pentium	**1993**	**3.1M**
Pentium/MMX	**1997**	**4.5M**

- Added special collection of instructions for operating on 64-bit vectors of 1, 2, or 4 byte integer data.

Pentium Pro	**1995**	**6.5M**

- Added conditional move instructions.
- Big change in underlying microarchitecture.

x86 Evolution: Programmer's View

Name	Date	Transistors
Pentium III	**1999**	**8.2M**

- Added "streaming SIMD" instructions for operating on 128-bit vectors of 1, 2, or 4 byte integer or FP data.

Pentium 4	**2001**	**42M**

- Added 8-byte formats and 144 new instructions for streaming SIMD.
- "Superpipelined" with very fast clocks.

Pentium 4 Xeon	**2003**	**125M**

- Added hyperthreading, large caches.

Pentium M	**2003**	**77M**

- Added hyperthreading, lower power.

x86 Evolution: Programmer's View

Name	Date	Transistors
Pentium 4 EE	**2005**	**164M**

- Includes hyperthreading.

Pentium EE 840	**2005**	**230M**

- Dual core, shared L2 cache.

Pentium EE 840	**2005**	**675M**

- 8 Mbyte L3 Cache.

Pentium Core Duo	**2006**	**250M+**

- Dual core, shared L2 cache.

"Nehalem" Core	**2008**	**731M**

- Added 256-bit media instructions, on-chip memory controller.

x86 Evolution: 64-bit

"Tick-tock" implementation strategy.

- Change processor in middle of stable technology.
- Change technology in middle of stable design.

Pentium i3, i5, i7 2010 1.16B

- "Sandy Bridge" 4 core, 2.27B transitors, 8 core.

Pentium i3, i5, i7 2012 1.4B

- "Ivy Bridge" Tri-gate (3-D) transistor technology.

New Species: IA64

Itanium **2001** **25M**

- Extends to IA64, a 64-bit architecture
- Radically new instruction set designed for high performance.
- Able to run existing IA32 programs with on-board "x86 engine."
- Joint project with Hewlett-Packard.

Itanium 2 **2002** **221M**

- Big performance boost.

Itanium 2, Big **2006** **410M**

- 24 Mbyte cache, good for server operations, expensive ($500 to $4000) per processor.

Not good market acceptance; Intel ended support.

x86 Evolution: Clones

Transmeta

Radically different approach to implementation.

- Translate x86 code into "very long instruction word" (VLIW) code.
- Very high degree of parallelism.

Centaur / Via

- Continued evolution from Cyrix, the 3rd x86 vendor. Low power, design team in Austin.
- 32-bit processor family.
  - At 2 GHz, around 2 watts; at 600 MHz around 0.5 watt.
- 64-bit processor family, used by HP, Lenovo, OLPC, IBM.
  - Very low power, only a few watts at 1.2 GHz.
  - Full virtualization and SSE support.

x86 Evolution: Clones

Advanced Micro Devices (AMD)

Historically,

- AMD has followed just behind Intel.
- A little bit slower, but enough cheaper to be competitive.

Recently,

- Recruited top circuit designers from Digital Equipment Corp.
- Exploited the fact that Intel was distracted by IA64.
- Now is a close competitor to Intel.

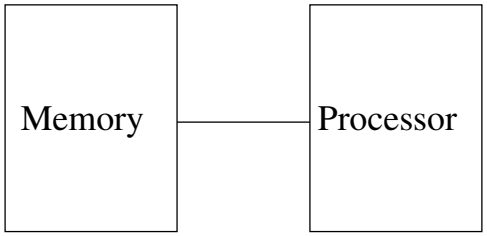
Developed first commerically-available x86 extension to 64-bits, which forced Intel to do the same.

Current,

- Delivering 64-bit x86 processors for desktop and server market.
- Have sophisticated point-to-point IO bus.
- Providing competitive products in desktop and server market.

Machine Models

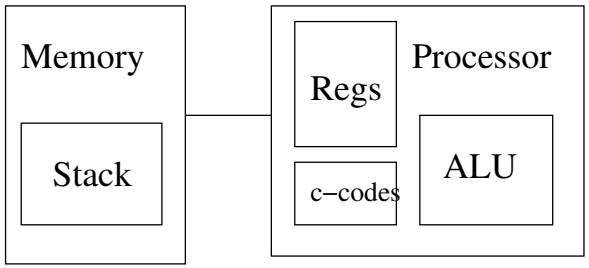
C



Data
1) char
2) int, float
3) double
4) struct, array
5) pointer

Control
1) loops
2) conditionals
3) switch
4) proc. call
5) proc. return

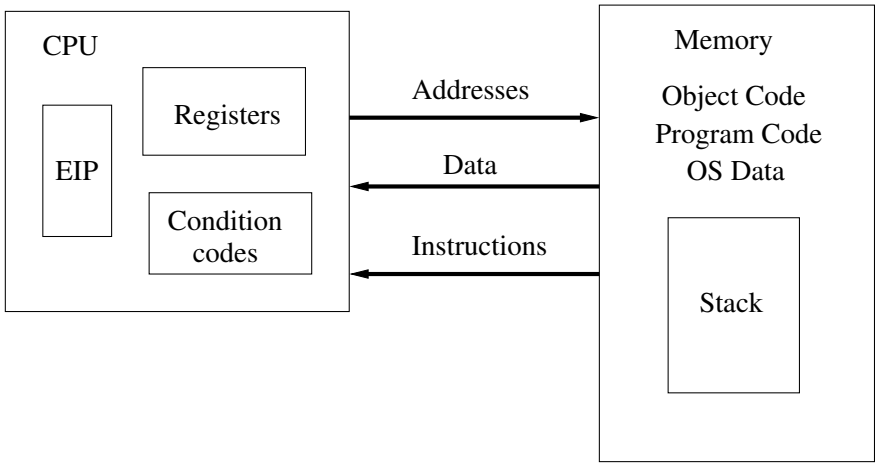
Assembly



1) byte
2) 2-byte word
3) 4-byte long word
4) contiguous byte allocation
5) address of initial byte

1) branch/jump
2) call
3) ret

Assembly Programmer's View



Programmer Visible State

- EIP (Program Counter): address of next instruction.
- Register file: heavily used program data.
- Condition codes:
  - Store status info about most recent arithmetic operation.
  - Used for conditional branching.

Memory

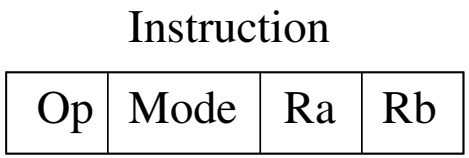
- Byte addressable array.
- Code, user data, (some) OS data.
- Includes stack.

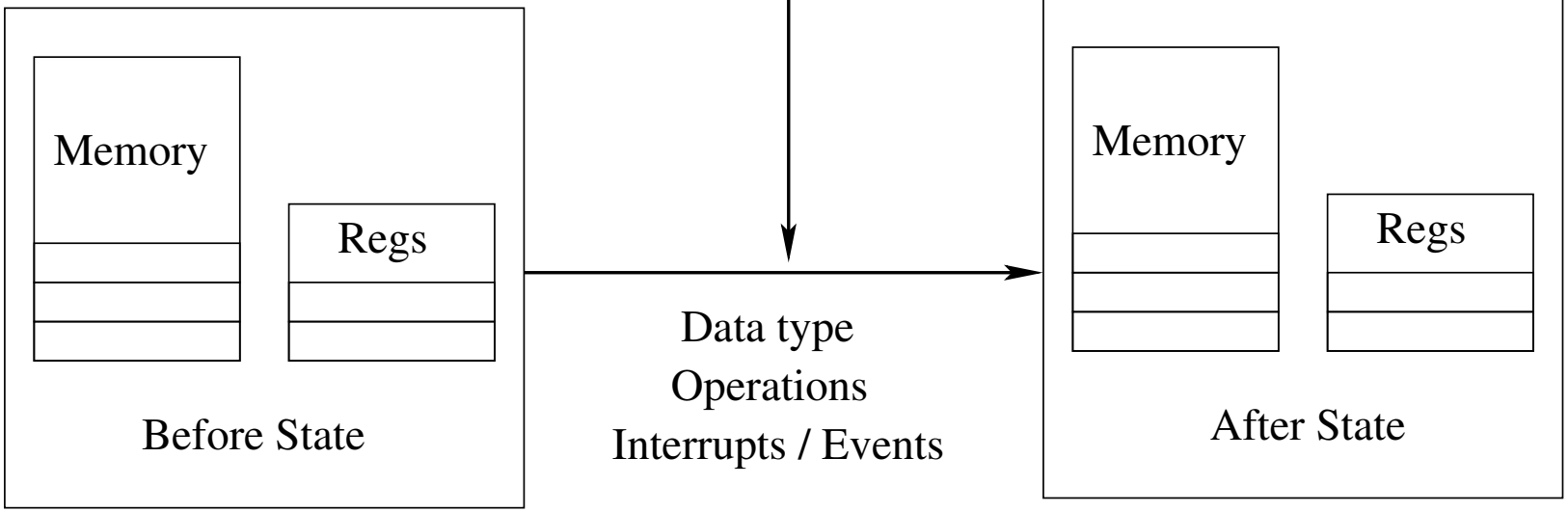
ISA Principles

- Contract between programmer and the hardware.
  - Defines visible state of the system.
  - Defines how state changes in response to instructions.
- For Programmer: ISA is model of how a program will execute.
- For Hardware Designer: ISA is formal definition of the correct way to execute a program.
  - With a stable ISA, SW doesn't care what the HW looks like under the hood.
  - Hardware implementations can change drastically.
  - As long as the HW implements the same ISA, all prior SW should still run.
  - Example: x86 ISA has spanned many chips; instructions have been added but the SW for prior chips still runs.
- ISA specification: the binary encoding of the instruction set.

ISA Basics

Instruction formats
Instruction types
Addressing modes





Machine State
Memory organization
Register organization

Architecture vs. Implementation

Architecture: defines what a computer system does in response to a program and set of data.

- Programmer visible elements of computer system.

Implementation: defines how a computer does it.

- Sequence of steps to complete operations.
- Time to execute each operation.
- Hidden "bookkeeping" function.

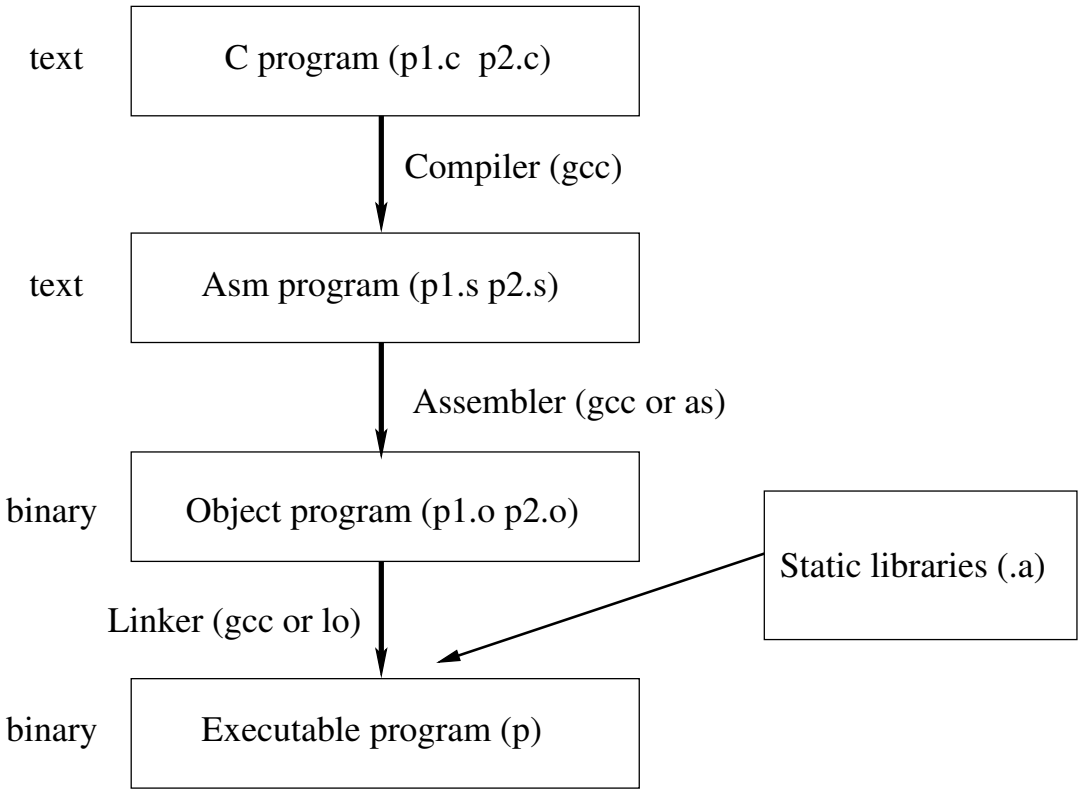
Examples

Architecture or Implementation?

- Number of general purpose registers
- Width of memory bus
- Binary representation of the instruction: `sub r4, r2, #27`
- Number of cycles to execute a FP instruction
- Which condition code bits are set on a `move` instruction
- Size of the instruction cache
- Type of FP format

Turning C into Object Code

- Code in files: `p1.c, p2.c`
- Compile with command: `gcc -O p1.c p2.c -o p`
- Use optimization (`-O`)
- Put resulting binary in file `p`



Compiling into Assembly

C Code:

```
int sum( int x, int y )
{
    int t = x+y;
    return t;
}
```

Run command: `gcc -O -S code.c`
produces file `code.s`.

```
_sum:
    pushl %ebp
    movl  %esp,%ebp
    movl  12(%ebp),%eax
    addl  8(%ebp),%eax
    movl  %ebp,%esp
    popl  %ebp
    ret
```

Assembly Characteristics

Minimal Data Types

- "Integer" data of 1, 2 or 4 bytes
- Addresses (untyped pointers)
- Floating point data of 4, 8 or 10 bytes
- No aggregate types such as arrays or structures
- Just contiguously allocated bytes in memory

Primitive Operations

- Perform arithmetic functions on register or memory data
- Transfer data between memory and register
- Load data from memory into register
- Store register data into memory
- Transfer control
- Unconditional jumps to/from procedures
- Conditional branches

Object Code

```
0x401040: <sum>:
    0x55
    0x89      # total of
    0xe5      # 13 bytes
    0x8b
    0x45      # each inst
    0x0c      # 1, 2, or
    0x03      # 3 bytes
    0x45
    0x08      # starts at
    0x89      # addr
    0xec      # 0x401040
    0x5d
    0xc3
```

Assembler

- Translates .s into .o
- Binary encoding of each inst.
- Nearly complete image of executable code
- Missing linkages between code in different files

Linker

- Resolves references between files
- Combines with static run-time libraries
- E.g., code for `malloc`, `printf`
- Some libraries are dynamically linked
- Linking just before execution.

Machine Instruction Example

```
int t = x + y;
```

```
    addl   8(%ebp),%eax
```

Similar to expression: `x += y`.

```
0x401046:      03 45 08
```

C Code

- Add two signed integers.

Assembly

- Add two 4-byte integers
- "Long" words in GCC terms
- Same instruction signed or unsigned
- Operands:
  - x: Register `%eax`
  - y: Memory `M[%ebp+8]`
  - t: Register `%eax`
- Return value in `%eax`

Object Code

- 3-byte instruction
- Stored at address `0x401046`

Disassembling Object Code

```
00401040 <_sum>:
   0:    55                push   %ebp
   1:    89 e5             mov    %esp, %ebp
   3:    8b 45 0c          mov    0xc(%ebp), %eax
   6:    03 45 08          add    0x8(%ebp), %eax
   9:    89 ec             mov    %ebp, %exp
   b:    5d                pop    %ebp
   c:    c3                ret
   d:    8d 76 00          lea    0x0(%esi), %esi
```

Disassembler

- `objdump -d p`
- Useful tool for examining object code
- Analyzes bit pattern of series of instructions
- Produces approximate rendition of assembly code
- Can be run one either a.out (complete executable) or .o file

Alternate Disassembly

Object code:

```
0x401040:
    0x55
    0x89
    0xe5
    0x8b
    0x45
    0x0c
    0x03
    0x45
    0x08
    0x89
    0xec
    0x5d
    0xc3
```

```
0x401040 <sum>:      push %ebp
0x401041 <sum+1>:    mov  %esp,%ebp
0x401043 <sum+3>:    mov  0xc(%ebp),%eax
0x401046 <sum+6>:    add  0x8(%ebp),%eax
0x401049 <sum+9>:    mov  %ebp,%esp
0x40104b <sum+11>:   pop  %ebp
0x40104c <sum+12>:   ret
0x40104d <sum+13>:   lea  0x0(%esi),%esi
```

Disassemble procedure:

```
x/13b sum
```

Examine the 13 bytes starting at sum.

Within gdb:

```
gdb p
disassemble sum
```

What Can be Disassembled?

- Anything that can be interpreted as executable code.
- Disassembler examines bytes and reconstructs assembly source.

```
% objdump -d WINWORD.EXE

WINWORD.EXE:     file format pei-i386

No symbols in "WINWORD.EXE".
Disassembly of section .text:

30001000 <.text>:
30001000:   55                      push %ebp
30001001:   8b ec                   mov  %esp, %ebp
30001003:   6a ff                   push $0xffffffff
30001005:   68 90 10 00 30          push $0x30001090
3000100a:   68 91 dc 4c 30          push $0x304cdc91
```

Whose Assembler?

Intel/Microsoft Format

```
lea   eax, [ecx+ecx*2]
sub   esp, 8
cmp   dword ptr [ebp-8], 0
mov   eax, dword ptr [eax
    *4+100h]
```

GAS/Gnu Format

```
leal (%ecx,%ecx,2), %eax
subl $8,%esp
cmpl $0,-8(%ebp)
movl $0x100(,%eax,4),%eax
```

Intel/Microsoft Differs from GAS

- Operands are listed in opposite order:

  `mov` Dest, Src `movl` Src, Dest

- Constants not preceded by '$'; denote hex with 'h' at end.

  `100h` `$0x100`

- Operand size indicated by operands rather than operator suffix.

  `sub` `subl`

- Addressing format shows effective address computation.

  `[eax*4+100h]` `$0x100(,%eax,4)`

Moving Data

From now on we'll always use GAS assembler format.

Moving Data:

- Form: `movl` Source, Dest
- Move 4-byte "long" word
- Lots of these in typical code

Operand Types

- Immediate: Constant integer data
  - Like C constant, but prefixed with '$'
  - E.g., `$0x400`, `$-533`
  - Encoded with 1, 2, or 4 bytes
- Register: One of 8 integer registers
  - But `%esp` and `%ebp` are reserved for special use
  - Others have special uses for particular instructions
- Memory: source/dest is first address of block (4 bytes for an int).
  - Various "addressing modes"

Registers

IA32 uses the same 8 registers as Y86.

%eax
%edx
%ecx
%ebx
%esi
%edi
%esp
%ebp

Some have addressable internal structure; more on that later.

movl Operand Combinations

Unlike the Y86, we don't distinguish the operator depending on the operand addressing modes.

Source	Dest.	Assembler	C Analog
Immediate	Register	`movl $0x4,%eax`	`temp = 0x4;`
Immediate	Memory	`movl $-147,(%eax)`	`*p = -147;`
Register	Register	`movl %eax,%edx`	`temp2 = temp1;`
Register	Memory	`movl %eax,(%edx)`	`*p = temp;`
Memory	Register	`movl (%eax),%edx`	`temp = *p`

Memory–memory transfers are not allowed within a single instruction.

Simple Addressing Modes

- **Register:** `Reg[R]`

```
movl  %ecx, %ebx
```

- **Normal (R):** `Mem[Reg[R]]`
  - Register R specifies memory address.
  - This is often called *indirect* addressing.

```
movl (%ecx), %eax
```

- **Displacement D(R):** `Mem[Reg[R]+D]`
  - Register R specifies start of memory region.
  - Constant displacement D specifies offset

```
movl 8(%ecb),%edx
```

Addresses and Pointers in C

C programming model is close to machine language.

- Machine language manipulates memory addresses.
  - For address computation;
  - To store addresses in registers or memory.
- C employs pointers, which are just addresses of primitive data elements or data structures.

Examples of operators * and &:

- `int a, b;` /* declare integers a and b */
- `int *a_ptr;` /* a is a pointer to an integer */
- `a_ptr = a;` /* illegal, types don't match*/
- `a_ptr = &a;` /* a_ptr holds address of a */
- `b = *a_ptr;` /* dereference a_ptr and assign value to b */

Using Simple Addressing Modes

```
void swap( int *xp, int *yp)
{
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

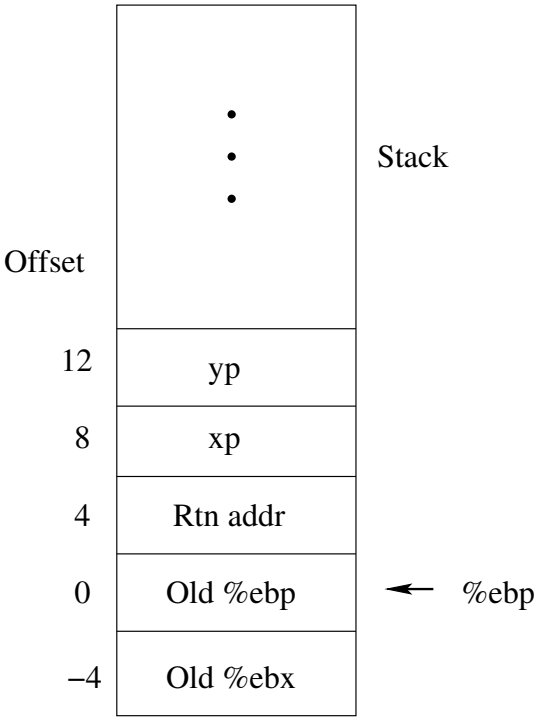
```
swap:
    pushl %ebp
    movl  %esp,%ebp
    pushl %ebx

    movl  12(%ebp),%ecx
    movl  8(%ebp),%edx
    movl  (%ecx),%eax
    movl  (%edx),%ebx
    movl  %eax,(%edx)
    movl  %ebx,(%ecx)

   # the following is
        tricky:
    movl  -4(%ebp),%ebx
    movl  %ebp,%esp
    popl  %ebp
    ret
```

Understanding Swap

Register	Variable
%ecx	yp
%edx	xp
%eax	t1
%ebx	t0



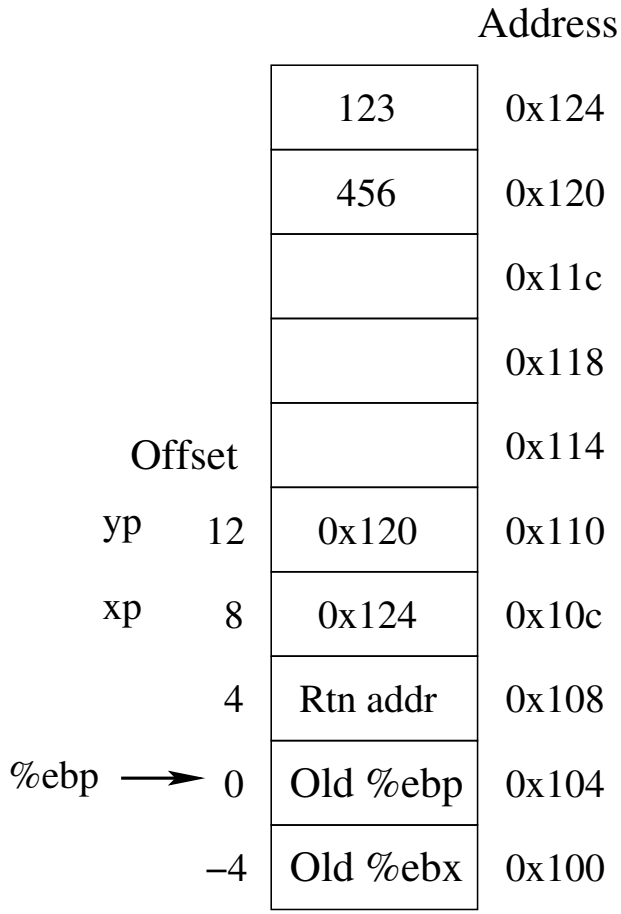

```
void swap( int *xp, int *yp)
{
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

```
movl   12(%ebp),%ecx # ecx = yp
movl   8(%ebp),%edx  # edx = xp
movl   (%ecx),%eax   # eax = *yp
movl   (%edx),%ebx   # ebx = *xp
movl   %eax,(%edx)   # *xp = eax
movl   %ebx,(%ecx)   # *yp = ebx
```

Understanding Swap (2)

```
movl   12(%ebp),%ecx  # ecx = yp
movl   8(%ebp),%edx   # edx = xp
movl   (%ecx),%eax    # eax = *yp
movl   (%edx),%ebx    # ebx = *xp
movl   %eax,(%edx)    # *xp = eax
movl   %ebx,(%ecx)    # *yp = ebx
```

%eax	
%edx	
%ecx	
%ebx	
%esi	
%edi	
%esp	
%ebp	0x104

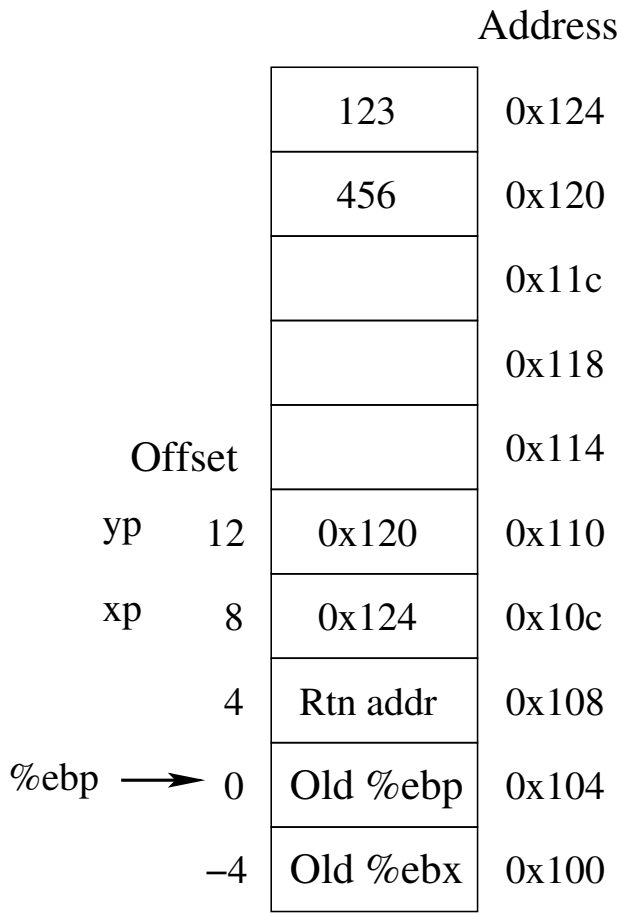


Understanding Swap (3)

```
movl   12(%ebp),%ecx # ecx = yp

movl   8(%ebp),%edx  # edx = xp
movl   (%ecx),%eax   # eax = *yp
movl   (%edx),%ebx   # ebx = *xp
movl   %eax,(%edx)   # *xp = eax
movl   %ebx,(%ecx)   # *yp = ebx
```

%eax	
%edx	
%ecx	0x120
%ebx	
%esi	
%edi	
%esp	
%ebp	0x104



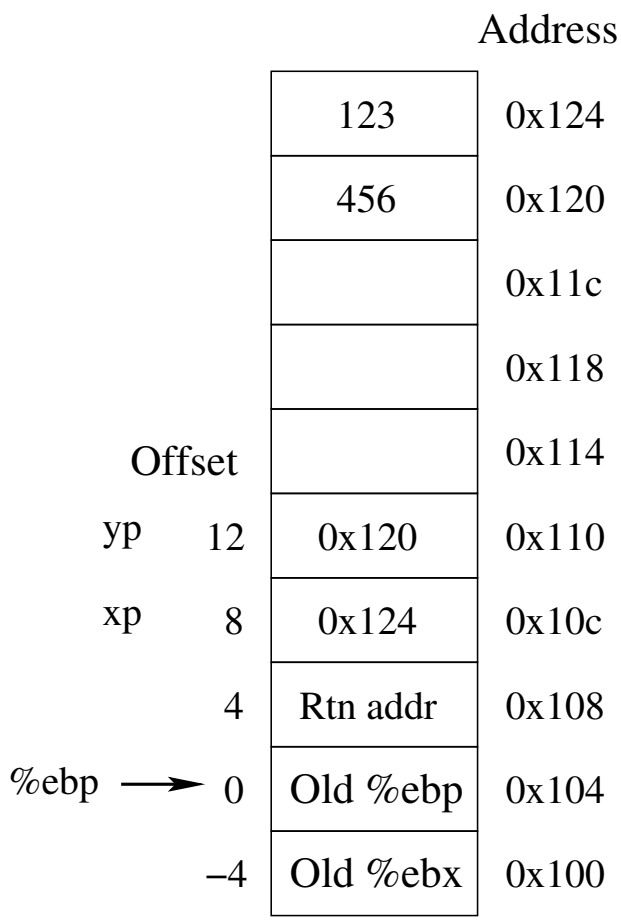
Understanding Swap (4)

```
movl   12(%ebp),%ecx  # ecx = yp

movl   8(%ebp),%edx   # edx = xp

movl   (%ecx),%eax    # eax = *yp
movl   (%edx),%ebx    # ebx = *xp
movl   %eax,(%edx)    # *xp = eax
movl   %ebx,(%ecx)    # *yp = ebx
```

%eax	
%edx	0x124
%ecx	0x120
%ebx	
%esi	
%edi	
%esp	
%ebp	0x104



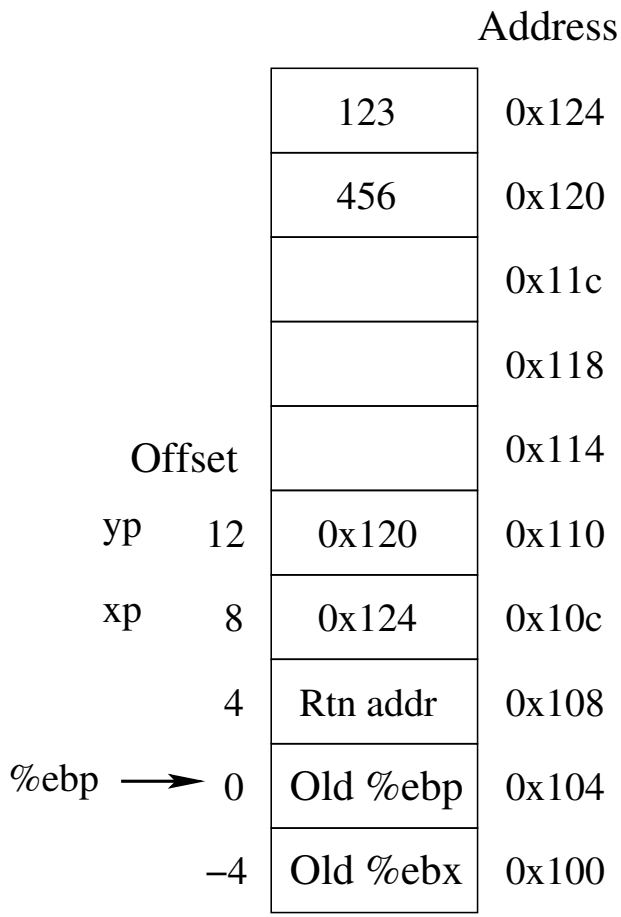
Understanding Swap (5)

```
movl   12(%ebp),%ecx  # ecx = yp
movl   8(%ebp),%edx   # edx = xp

movl   (%ecx),%eax    # eax = *yp

movl   (%edx),%ebx    # ebx = *xp
movl   %eax,(%edx)    # *xp = eax
movl   %ebx,(%ecx)    # *yp = ebx
```

%eax	456
%edx	0x124
%ecx	0x120
%ebx	
%esi	
%edi	
%esp	
%ebp	0x104



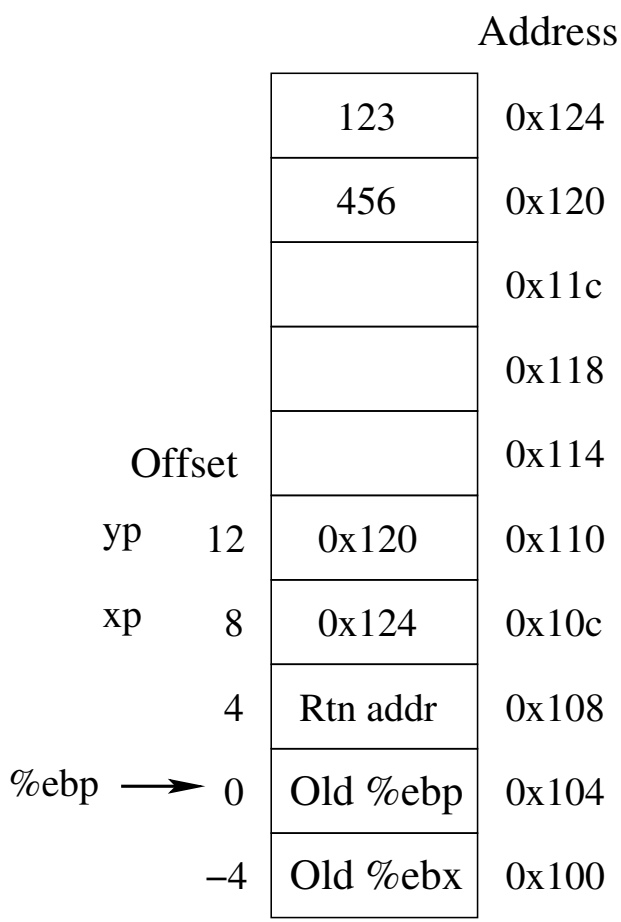
Understanding Swap (6)

```
movl   12(%ebp),%ecx  # ecx = yp
movl   8(%ebp),%edx   # edx = xp
movl   (%ecx),%eax    # eax = *yp

movl   (%edx),%ebx    # ebx = *xp

movl   %eax,(%edx)    # *xp = eax
movl   %ebx,(%ecx)    # *yp = ebx
```

%eax	456
%edx	0x124
%ecx	0x120
%ebx	123
%esi	
%edi	
%esp	
%ebp	0x104



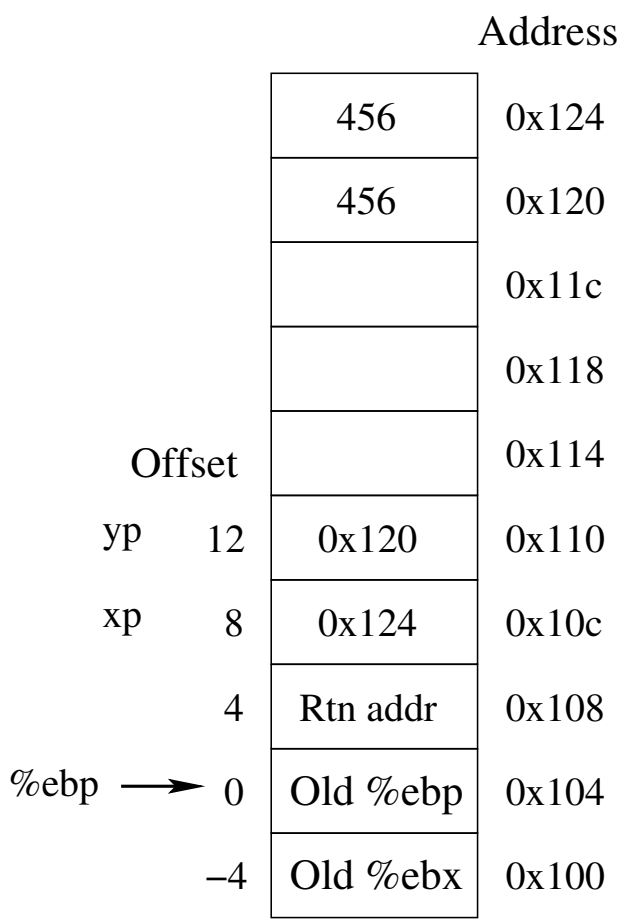
Understanding Swap (7)

```
movl   12(%ebp),%ecx  # ecx = yp
movl   8(%ebp),%edx   # edx = xp
movl   (%ecx),%eax    # eax = *yp
movl   (%edx),%ebx    # ebx = *xp

movl   %eax,(%edx)    # *xp = eax

movl   %ebx,(%ecx)    # *yp = ebx
```

%eax	456
%edx	0x124
%ecx	0x120
%ebx	123
%esi	
%edi	
%esp	
%ebp	0x104

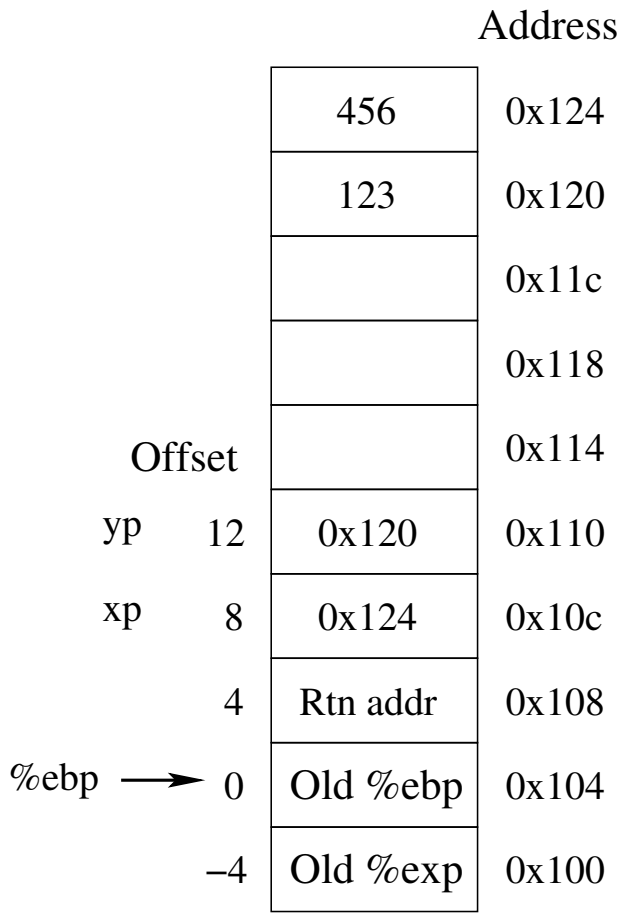


Understanding Swap (8)

```
movl   12(%ebp),%ecx  # ecx = yp
movl   8(%ebp),%edx   # edx = xp
movl   (%ecx),%eax    # eax = *yp
movl   (%edx),%ebx    # ebx = *xp
movl   %eax,(%edx)    # *xp = eax

movl   %ebx,(%ecx)    # *yp = ebx
```

%eax	456
%edx	0x124
%ecx	0x120
%ebx	123
%esi	
%edi	
%esp	
%ebp	0x104



Indexed Addressing Modes

Most General Form:

`D(Rb, Ri, S)`	`Mem[Reg[Rb] + S*Reg[Ri] + D]`

- D: Constant "displacement" of 1, 2 or 4 bytes
- Rb: Base register, any of the 8 integer registers
- Ri: Index register, any except `%esp` (and probably not `%ebp`)
- S: Scale, one of 1, 2, 4 or 8.

Special Cases:

`(Rb, Ri)`	`Mem[Reg[Rb] + Reg[Ri]]`
`D(Rb, Ri)`	`Mem[Reg[Rb] + Reg[Ri] + D]`
`(Rb, Ri, S)`	`Mem[Reg[Rb] + S * Reg[Ri]]`

Addressing Modes

Type	Form	Operand value	Name
Immediate	\$D	D	Immediate
Register	E_a	$R[E_a]$	Register
Memory	D	$M[D]$	Absolute
Memory	(E_a)	$M[R[E_a]]$	Indirect
Memory	$D(E_b)$	$M[D + R[E_b]]$	Base + displacement
Memory	(E_b, E_i),	$M[R[E_b] + R[E_i]]$	Indexed
Memory	$D(E_b, E_i)$,	$M[D + R[E_b] + R[E_i]]$	Indexed
Memory	$(, E_i, s)$	$M[R[E_i] \cdot s]$	Scaled indexed
Memory	$D(, E_i, s)$	$M[D + R[E_i] \cdot s]$	Scaled indexed
Memory	(E_b, E_i, s),	$M[R[E_b] + R[E_i] \cdot s]$	Scaled indexed
Memory	$D(E_b, E_i, s)$	$M[D + R[E_b] + R[E_i] \cdot s]$	Scaled indexed

The scaling factor s must be either 1, 2, 4, or 8.

Address Computation Example

%edx	0xf000
%ecx	0x100

Expression	Computation	Address
0x8(%edx)	0xf000 + 0x8	0xf008
(%edx, %ecx)	0f000 + 0x100	0xf100
(%edx, %ecx, 4)	0xf000 + 4*0x100	0xf400
0x80(,%edx, 2)	2*0xf000 + 0x80	0x1e080

Address Computation Instruction

Form: `leal  Src, Dest`

- Src is address mode expression.
- Sets Dest to address denoted by the expression

Uses:

- Computing address without doing a memory reference:
  - E.g., translation of `p = &x[i];`
- Computing arithmetic expressions of the form $x + k \times y$, where $i \in \{1, 2, 4, 8\}$

Some Arithmetic Operations

Two operand instructions:

Format	**Computation**	
`addl` Src, Dest	`Dest = Dest + Src`	
`subl` Src, Dest	`Dest = Dest - Src`	
`imull` Src, Dest	`Dest = Dest * Src`	
`sall` Src, Dest	`Dest = Dest << Src`	also called `shll`
`sarl` Src, Dest	`Dest = Dest >> Src`	arithmetic
`shrl` Src, Dest	`Dest = Dest >> Src`	logical
`xorl` Src, Dest	`Dsst = Dest ^ Src`	
`andl` Src, Dest	`Dest = Dest & Src`	
`orl` Src, Dest	`Dest = Dest | Src`	

Some Arithmetic Operations

One operand instructions:

Format	**Computation**
`incl` Dest	`Dest = Dest + 1`
`decl` Dest	`Dest = Dest - 1`
`negl` Dest	`Dest = -Dest`
`notl` Dest	`Dest = ~Dest`

Using leal for Arithmetic Expressions

```
int arith
    (int x, int y, int z)
{
    int t1 = x+y;
    int t2 = z+t1;
    int t3 = x+4;
    int t4 = y * 48;
    int t5 = t3 + t4;
    int rval = t2 * t5;
    return rval;
}
```

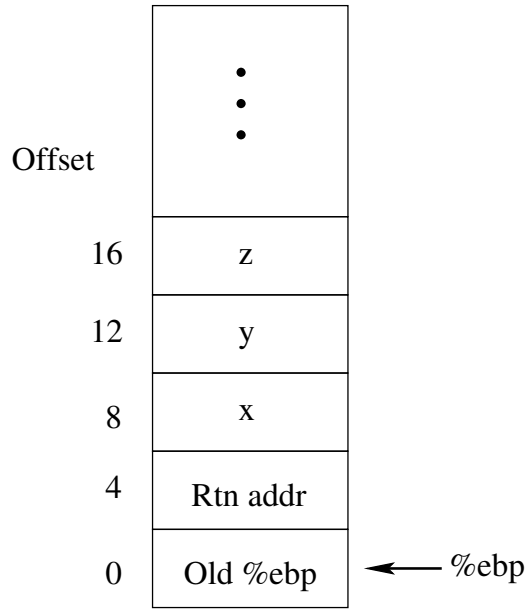
```
arith:
# set up
    pushl %ebp
    movl  %esp, %ebp

# body
    movl  8(%ebp), %eax
    movl  12(%ebp), %edx
    leal  (%edx,%eax),%ecx
    leal  (%edx,%edx,2),%edx
    sall  $4,%edx
    addl  16(%ebp),%ecx
    leal  4(%edx,%eax),%eax
    imull %ecx,%eax

# finish
    movl  %ebp,%esp
    popl  %ebp
    ret
```

Understanding Arithmetic

```
int arith
    (int x, int y, int z)
{
    int t1 = x+y;
    int t2 = z+t1;
    int t3 = x+4;
    int t4 = y * 48;
    int t5 = t3 + t4;
    int rval = t2 * t5;
    return rval;
}
```




```
    movl   8(%ebp), %eax          # eax = x
    movl   12(%ebp), %edx         # edx = y
    leal   (%edx,%eax),%ecx       # ecx = x+y (t1)
    leal   (%edx,%edx,2),%edx     # edx = 3*y
    sall   $4,%edx                # edx = 48*y (t4)
    addl   16(%ebp),%ecx          # ecx = z+t1 (t2)
    leal   4(%edx,%eax),%eax      # eax = 4+t4+x (t5)
    imull %ecx,%eax               # eax = t5*t2 (rval)
```

Understanding Arithmetic

```
int arith
    (int x, int y, int z)
{
    int t1 = x+y;
    int t2 = z+t1;
    int t3 = x+4;
    int t4 = y * 48;
    int t5 = t3 + t4;
    int rval = t2 * t5;
    return rval;
}
```

```
# eax = x
    movl   8(%ebp), %eax
# edx = y
    movl   12(%ebp), %edx
# ecx = x+y (t1)
    leal   (%edx,%eax),%ecx
# edx = 3*y
    leal   (%edx,%edx,2),%edx
# edx = 48*y (t4)
    sall   $4,%edx
# ecx = z+t1 (t2)
    addl   16(%ebp),%ecx
# eax = 4+t4+x (t5)
    leal   4(%edx,%eax),%eax
# eax = t5*t2 (rval)
    imull %ecx,%eax
```

Another Example

```
int logical( int x, int y)
{
    int t1 = x^y;
    int t2 = t1 >> 17;
    int mask = (1<<13) - 7;
    int rval = t2 & mask;
    return rval;
}
```

```
logical:
  pushl %ebp
  movl  %esp,%ebp
  movl  8(%ebp),%eax
  xorl  12(%ebp),%eax
  sarl  $17,%eax
  andl  $8185,%eax
  movl  %ebp,%esp
  popl  %ebp
  ret
```

Note:
$2^{13} = 8192; 2^{13} - 7 = 8185$

```
  movl  8(%ebp),%eax     # eax = x
  xorl  12(%ebp),%eax    # eax = x^y (t1)
  sarl  $17,%eax         # eax = t1>>17 (t2)
  andl  $8185,%eax       # eax = t2 & 8185
```